CLAM: Concurrent Location Management for Moving Objects

Jing Dai
Department of Computer Science
Virginia Polytechnic Institute and State University
7054 Haycock Road, Falls Church, VA 22043
daij@vt.edu

Chang-Tien Lu
Department of Computer Science
Virginia Polytechnic Institute and State University
7054 Haycock Road, Falls Church, VA 22043
ctlu@vt.edu

ABSTRACT
Recently, with the broad usage of location-aware devices, applications with moving object management became very popular. In order to manage moving objects efficiently, many spatial/spatial-temporal data access methods have been proposed. However, most of these data access methods are designed for single-user environments. In multiple-user systems, frequent updates may cause a significant number of read-write conflicts using these data access methods. In this paper, we propose an efficient framework, Concurrent LocAtion Management (CLAM), for managing moving objects in multiple-user environments. The proposed concurrency control protocol integrates the efficiency of the link-based approach and the flexibility of the lock-coupling mechanism. Based on this protocol, concurrent location update and search algorithms are provided. We formally analyze and prove the correctness of the proposed concurrent operations. Experiment results on real datasets validate the efficiency and scalability of the proposed concurrent location management framework.

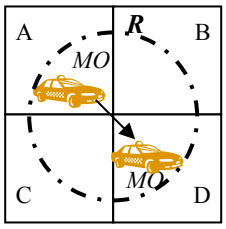
Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *concurrency, query processing.*

Keywords
Spatial database, Concurrency control, Space filling curve, B-tree.

1 INTRODUCTION
Recently, with the broad usage of location-aware devices, applications with moving object management, such as vehicle management systems and emergency response systems, became very popular. In order to manage moving objects efficiently, many spatial/spatial-temporal data access methods have been proposed. One important category of these data access methods is based on B+-tree and Space-Filling Curve (SFC)[4]. Utilizing SFC to linearly map multidimensional data to one-dimensional space, thus retaining the usage of one-dimensional B+-tree, is a cost-effective solution compared to other spatial access methods, such as R-trees[2, 5] and grid files[17], because many maturely developed modules, e.g., query optimization, in the traditional database management systems can be reused. In this scenario, multidimensional access methods based on SFC have attracted many research efforts. Comparisons among different space-filling curves have been conducted based on their characteristics[15, 16], and several query algorithms have been proposed to support the operations on SFCs[11, 14, 18].

ACMGIS'07, November 7-9, 2007, Seattle, WA
Copyright 2007 ACM ISBN 978-1-59593-914-2/07/11...$5.00

Concurrency control is one of the important DBMS techniques. As stated in the Lowell Report [1], "We face major changes in the traditional DBMS areas, such as ..., **concurrency control**, ..., technology keeps changing the rules. These changing ratios require us to reassess storage management and query processing algorithms." Most of the data access methods for moving objects are designed for single-user environments, even though in multiple-user systems, frequent updates may cause a significant number of read-write conflicts using these access methods. For example, a location update for a moving object usually consists of one delete operation and one insert operation. If there happens to be another operation trying to access the moving object after its previous location was deleted, and before the new location is inserted, the system will fail to retrieve this object. We name this inconsistent situation as **pseudo disappearance**, which is a scenario that violates both serializabile isolation and data consistency rules[20]. An example of pseudo disappearance is shown in Figure 1, where the search operation fails to retrieve the moving object *MO*, even though *MO* actually exists within the search region *R*. A similar scenario may also occur among two update operations. To prevent this inconsistent status of pseudo disappearance from happening, concurrency control protocols have to be designed for managing these concurrent data accesses in multi-user environments.



Operations:
1. Update *MO* from *A* to *D*
2. Search in region *R*

Processing Sequence:
1. Update(*MO*): Delete from *A*
2. Search(*R*): Search *A,B,C,D*
3. Update*(MO)*: Insert into *D*

Search Result: Empty

Figure 1. An example of pseudo disappearance.

For B+-trees, many link-based concurrency control protocols have been proposed to utilize the global order of the entries in the leaf nodes of the index tree, and they have been shown to be efficient and easy to implement [9, 10, 12]. Since the SFC-based indexing methods use B+-trees as the underlying indices, an intuitive solution is to directly apply these link-based approaches to provide concurrency control. However, spatial operations require special considerations to assure the consistency of the operation results. For example, a spatial range query may contain multiple inconsecutive range queries on the B+-tree, therefore, it requires all of these B+-tree queries return the valid results at the committing point. Furthermore, some location updates may not require modifications on the index nodes of the B+-tree. In this case, requesting these unnecessary locks on the B+-tree can degrade the system throughput, whereas not requesting adequate locks may cause inconsistency. To prevent these concurrent spatial operations from interfering with each other, link-based approaches are not sufficient as they cannot protect multiple inconsecutive

ranges among updates, nor handle concurrent spatial updates efficiently. Therefore, a new protocol is demanded for spatial concurrency protection.

This paper proposes Concurrent LocAtion Management (CLAM), a framework of concurrent location update and query based on the multidimensional indexing structure with general SFCs and B^{link}-tree. This framework applies a concurrency control protocol which integrates the link-based approach with the lock-coupling mechanism. Spatial location update and range query algorithms based on the proposed protocol are designed. The major contributions of this paper are as follows:

- The proposed concurrent framework secures serializability and consistency without incurring additional deadlocks for concurrent spatial location updates and queries on multidimensional indexing structures with general SFCs and B^{link}-tree;
- The proposed concurrency control protocol preserves the simplicity and efficiency of the link-based approach, and adopts the flexibility of the lock-coupling method;
- A formal proof is provided to show the correctness of the proposed CLAM framework; and experiment results on benchmark datasets validate the efficiency and scalability of the proposed concurrent operations and framework.

This paper is organized as follows: Section 2 reviews multidimensional access methods using SFC and B-tree family, as well as the concurrency control solutions for the B-tree family; the concurrent spatial range query and location update operations are defined in Section 3; Section 4 provides the algorithms for concurrent spatial operations, followed by the correctness proof in Section 5; experiments are illustrated and analyzed in Section 6; and finally Section 7 gives the conclusion.

2 RELATED WORK AND MOTIVATION

Space-filling curves, as a linear mapping schema applied in spatial data access methods, have been extensively studied, e.g., Peano-Z curve[19] maps data from a unit interval to a unit square; Gray code curve[3] improves Z-code by hashing; Hilbert curve[6] generalizes the concept to a mapping of the whole space. A historical survey of SFCs is conducted in [21]. Among the different types of SFCs, Hilbert curve has been shown to preserve the premier data locality under most circumstances [7]. Recently, several major SFCs, including Hilbert, Peano-Z, Gray, Scan, and Sweep, are systematically compared [15, 16]. Integrated with SFC, the widely used B+-tree family can constitute efficient spatial data access methods, especially for processing frequently updated point data, e.g., moving objects, in practical applications.

Many concurrency control protocols [8-10, 12, 22] have been proposed to support general concurrent search and update operations on B+-trees. These protocols can be classified into two categories, namely, link-based and lock-coupling. The link-based approaches [9, 10, 12] employ only one type of locks in all operations, and their read operations do not require any locks. In these approaches, a lock-free read operation follows the links from the root to leaf to identify the search path level by level, and traverses rightward if the current node does not contain the queried key. Other operations in these approaches request locks without interfering with the read operations. A complete set of concurrent operations on B^{link}-trees, including read, insert, delete, modify, and reorganize, has been discussed in [9]. In this approach, the update operations (insert, delete, and modify) only place locks on leaf nodes, but not on internal nodes. The reorganize operation, which restructures the tree periodically, accesses the metadata of the index to merge or split internal nodes. This approach utilizes the left-to-right links between the nodes on the same level to construct a simple and efficient concurrency control protocol on semi-dynamic B^{link}-trees. To build symmetric concurrent B^{link}-trees, a two-phase merge algorithm has been proposed in [10] to prevent periodical reorganization.

In multidimensional environments, however, the link-based concurrency control is not sufficient to meet the requirement that the multiple inconsecutive B-tree searches contained in one spatial range query occur at one time. In other words, it cannot guarantee the freshness of search results from interfering with overlapped update operations. The other category, lock-coupling approaches[8, 22], applies at least two types of locks, read-lock and write-lock, to provide flexible concurrency control. These distinct locks are combined to assure that the B-tree is correctly and dynamically modified, and help data recovery for aborted transactions. Since it is difficult to define a global order for spatial objects, the R-tree family generally adopts the lock-coupling approaches. Lock-coupling approaches can provide more flexible protection than the link-based protocols, albeit require complex lock maintenance on different levels of the index tree.

In the proposed CLAM framework, we incorporate the flexibility of the lock-coupling approach and the efficiency of the link-based mechanism to construct a new concurrency control protocol, designed specifically for spatial data access using B^{link}-trees and SFCs.

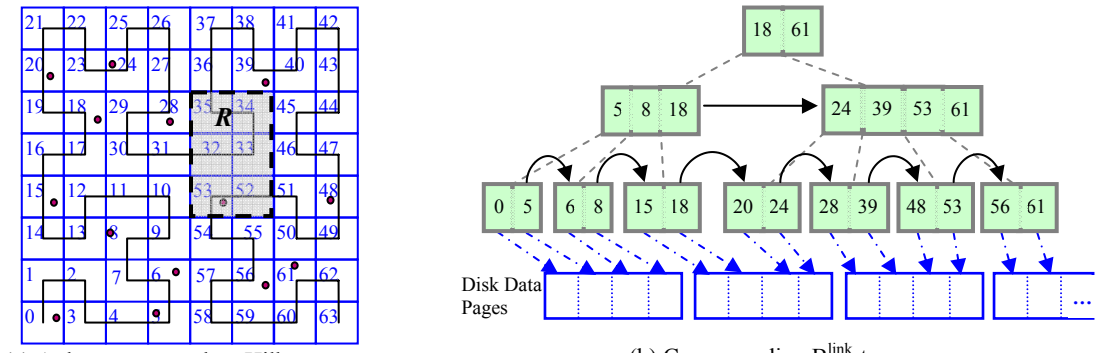
3 PRELIMINARY

3.1 Problem Formulation

In order to propose appropriate solutions for the concurrent location update problem, the application environment has to be described. In this problem, the data access method is based on SFCs and B^{link}-trees (a variant of the B+-tree, as shown in Figure 2)[12]. All the multidimensional data are mapped into equal-sized cells via SFC, associated with their corresponding cell *ID*s. The resolution of grid cells can be determined by the data distribution and disk page size. Then a B^{link}-tree is used to index these SFC values and to maintain the pointers to the data pages. The location management operations based on this data access method are defined as the following.

The execution of a **range query** returns a set of data objects that overlaps with a given range at the time the query is finished. Formally specified, the input of a range query is a *d*-multidimensional region (query window) *R*, a *d*-dimensional dataset *S*, and the corresponding spatial index *I*. The output of this range query is a set of data points (o_1, o_2, ..., o_n) covered in *R* within *S*. The output data should be valid at the committing time.

A **location update** operation inputs both the old position and new location of a *d*-dimensional data point *o*, as well as a *d*-dimensional dataset *S* and the corresponding spatial index *I*, and outputs the updated *I* and *S*. This operation contains two sub-tasks: delete the old position and insert the new location. An **insert** operation will add a new *d*-dimensional data point. A **delete** operation will remove an existing data point. Both need to update the index if necessary. The execution of a location update should not affect the results of a query before this update finishes, but



(a) A dataset mapped on Hilbert curve. (b) Corresponding B^{link}-tree.

Figure 2. A point dataset with Hilbert curve mapping and the corresponding B^{link}-tree.

have to be reflected in the results of queries after this update commits.

Several assumptions are made as follows to give detailed descriptions to this problem.

1. The multidimensional dataset contains only points, which means each data record is treated as a single point in a multidimensional space.
2. The space-filling curves are applied to provide the global cell order, and a standard B^{link}-tree is used as the one-dimensional access method (See Figure 2). Each leaf node of the B^{link}-tree contains the cell *ID*s and the pointers to the data pages that store the corresponding objects.
3. The read/write operation of a node from/to disk is an atomic action. And there exists a lock management module to maintain the requested locks and check the operation compatibility.

These assumptions are reasonable in real spatial applications, and they add constraints to the problem formulation.

The goal of the concurrency control protocol is to achieve serializable isolation and data consistency. Serializable isolation means that concurrent operations can acquire the same results as they are sequentially and separately executed. Data consistency refers to the requirement that the query can securely retrieve the data consistent to the valid database state.

3.2 Observations

The data access model based on SFC and B-tree family leaves spaces for efficiency improvement of concurrent spatial operations. Observations regarding the protocol design and potential performance improvements are discussed as follows.

Concurrency Control on SFC - In the SFC-based spatial index, the reorganization of B^{link}-trees occurs when inserting a new object to an empty cell, or when one cell becomes empty. Concurrency control is essential for multiple-user environments because updating either index tree nodes or data pages needs exclusive access to keep the data and query results consistent. The link-based concurrency control for B^{link}-trees inserts and deletes entries in a way that the reader can always fetch the valid data by following the left-to-right links to the node with the corresponding key range without placing any locks [9]. This approach is efficient, but cannot be directly applied for spatial queries. Because one such spatial query may require multiple inconsecutive searches on the B^{link}-tree, a secure protection mechanism has to be devised to keep the searched area unaltered until all the searched objects are returned. For example, the range query *R* in Figure 2(a) covers cell 32, 33, 34, 35, 52, and 53. It requires two B^{link}-tree range searches, cell cluster 32 to 35 and cell cluster 52 to 53. Therefore, not only the indexed cell 53, but also all the other five empty cells (32, 33, 34, 35, and 52) require to be protected from other update operations. Apparently, the link-based approach is not adequate to assure this kind of protection, as it can only protect the indexed cells among updates. Therefore, lock-coupling techniques have to be incorporated. Once the update operation needs to insert/delete a cell or modify a data object, it has to assure that the cell/object is not located within the ongoing search region. Note that not only the non-empty cells, but also the empty cells in the search area have to be locked, because in case a new object is inserted into an empty cell within the search range, it will tarnish and invalidate the final search results.

Lock Efficiency - In the spatial access method based on B^{link}-trees and SFCs, location update operations can be handled in different manners based on the SFC cells that contain the old location or new location. The index tree modification only occurs when inserting a new object to an empty cell, or when a cell becomes empty (taking Figure 1 as an example, cell *D* is empty, or cell *A* has only one object before the update). Otherwise, only the corresponding data pages need to be protected and modified. Obviously, if these distinct scenarios are not handled separately, the concurrency control protocol will need to lock all the B^{link}-tree leaf nodes and data cells that will be accessed during the update, and release them in the very end, which could significantly degrade the system throughput. Therefore, for performance consideration, these scenarios need to be respectively treated, so that unnecessary locks can be quickly released to increase the concurrency level.

4 CONCURRENT SPATIAL OPERATIONS

To explain the algorithms for concurrent operations in CLAM, the fundamental locking mechanism will be illustrated in the following subsection, followed by the detailed location update operation and range query algorithms.

4.1 Lock Map

In the proposed concurrent spatial operations in CLAM, two levels of locks are used (shown in Table 1). One is **node-level lock**, which is placed on index tree nodes. The node-level locks, requested only by update operations, are all write-locks. The other is **cell-level lock**, which includes both read-lock and write-lock and will be requested on SFC cells by all the spatial operations. As discussed in Section 3, for spatial queries, not only non-empty cells, but also empty cells will need to be read-locked, since they

are not allowed to be modified during the query process. Therefore, an auxiliary **lock map** structure, in addition to the index tree, is applied in the proposed framework to dynamically maintain cell-level locks. The concurrent spatial operations have to check the corresponding records in the lock map before placing cell-level locks. Each cell maintains a counter for its current read-locks, and use -1 to indicate the write-lock. A queue is also used by each cell to store the pending cell-level locks, so that these waiting processes can be awaked once the cell is available. After an operation checks the compatibility of the cell-level locks, if the cell is currently unavailable (i.e., has incompatible locks), this operation will be recorded in the pending queue. A lock map example is illustrated in Figure 3, where the pending queues of two cells are shown. As the lock map will be frequently accessed in this concurrency control framework, it can be implemented using a hash table that is evenly sliced based on the spatial distribution to avoid causing a performance bottleneck. In addition, sophisticated compression techniques can be applied to reduce the space requirement. Note that only cell-level locks need to access the lock map, because a cell-level lock will not conflict with a node-level lock in CLAM.

Table 1. Types of locks and their compatibility in CLAM.

		Cell-level		Node-level
		Write-lock	Read-lock	Write-lock
Cell-level	Write-lock	*Exclusive*	*Exclusive*	*N/A*
	Read-lock	*Exclusive*	*Compatible*	*N/A*
Node-level	Write-lock	*N/A*	*N/A*	*Exclusive*

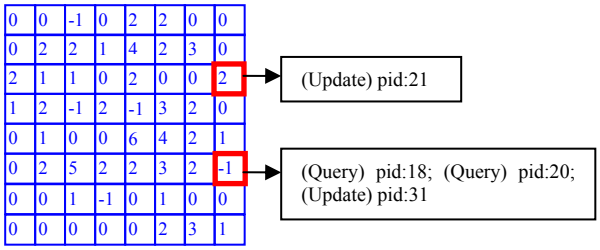


Figure 3. A lock map example.

4.2 Location Update

In order to protect the search operations from the interference of update operations, the location update operations need to check both the node-level locks on the B^{link}-tree leaf nodes, and the cell-level locks on the data cells. On the other hand, the query operations have to check the write-locks on the cells. For cell-level locks, the write-lock from update operations and the read-lock from read operations are exclusive to each other. In case the cells for updating have been locked by another operation, the update operations on these cells have to wait until they can successfully write-lock them. Specifically, the concurrent location update is performed as follows.

A location update operation first deletes an existing object, and then inserts a new object with the same object identifier (*ID*) to the data page. In this operation, there will be two exclusive scenarios: (1) the old location is not the last item in its cell, and the new location is located in a cell corresponding to an entry in the B^{link}-tree, thus the index tree will not need to be modified; (2) the new location is located in a cell that is not indexed in the B^{link}-tree (i.e., an empty cell), or, the old location is the last item in its cell, thus the corresponding nodes in the B^{link}-tree have to be locked and modified.

To perform an update operation, the **first phase, identification,** is to locate the corresponding leaf node that contains or will contain the cell of the new location, as well as the leaf node that contains the cell of the old location. To pinpoint the leaf nodes, the SFC values of the locations are calculated based on the specific curve. Then the B^{link}-tree is traversed to find the entry corresponding to the cell, in a similar way as the fundamental read operation on B^{link}-trees (as introduced in Section 2), except that this process needs to cache the traversed path. The cached path can help locate the parent nodes in case of node split or merge. By the end of this identification phase, node-level locks are requested on the leaf nodes that have been located.

Algorithm Location Update (*old_loc, new_loc, T, LM*)
Input: *old_loc:* location to be removed, *new_loc*: location to be inserted, *T:* B^{link}-tree, *LM:* Lock map,
Output: *T:* Updated B^{link}-tree.

//Identification
1. *c_old = SFC_map(old_loc)*; //determine the cell contains *old_loc*
2. *c_new = SFC_map(new_loc)*; //determine the cell contains *new_loc*
3. *n_old = T.traverse(c_old)*; // locate the leaf which contains *c_old*
4. *n_new = T.traverse(c_new)*; // locate the leaf which contains *c_new*
5. *T.writeLock(n_new & n_old);* // request node-level locks at one time

//Modification
6. *LM.writeLock(c_old* and *c_new)*; // request write-lock on cell *c_old* and *c_new*
7. If (*c_new.size > 0*) // *c_new* is not empty
8. *T.unWriteLock(n_new)*; // release lock on node *n_new*
9. If (*c_old.size > 1* or *c_old == c_new*) // no need to delete *c_old*
10. *T.unWriteLock(n_old);*// release lock on node *n_old*
11. If (*c_old.size == 1* and *c_old != c_new*) // need to delete *c_old*
12. *n_old.removeEntry(c_old);* //delete entry for *c_old* from *n_old*
13. If (*n_old.underflow == true*)
14. *n_old.merge()*;
15. *T.unWriteLock(n_old)*; // release node-level write-lock on *n_old*
16. If (*c_new.size = 0*) // *c_new* is empty
17. *n_new.addEntry(c_new)*; //add entry for *c_new* into node *n_new*
18. If (*n_new.overflow == true*)
19. *n_new.split()*;
20. *T.unWriteLock(n_new);*// release node-level write-lock on *n_new*
21. *PageDeletet(n_old.entry(c_old), old_loc)*; //remove *old_loc* from the data page that contains cell *c_old*
22. *PageInsert(n_new.entry(c_new), new_loc)*; //insert *new_loc* to the data page that contains cell *c_new*

//Commitment
23. *LM.unWriteLock(c_old* and *c_new)*; // remove cell-level write-locks
24. Return *T*;

Algorithm 1. Concurrent location update.

The **second phase, modification,** is to access the actual data page and update its content. This phase contains two conditional branches, corresponding to the two scenarios determined by the number of objects in the data cells. The **first** branch will request write-locks on the cells to be modified at one time. The request of the write-locks needs to access the lock map to determine whether this process should continue or be suspended. If the cell in the lock map has the value 0 (i.e., not locked), this operation will mark it as -1 (write-locked) and continue; otherwise, it will enqueue its process *ID* and pend. After locking the corresponding SFC cells, the node-level locks requested in the identification phase will be released, since no nodes will be modified. The algorithm will then update the data pages by inserting the new location and deleting the old position. The **second** branch, which occurs when the object moves from a single-item cell or relocates to an empty cell, requires keeping node-level locks on the B^{link}-tree. In this branch,

the cells that contain the new or old location will be write-locked, and then the leaf node that does not need to be changed, if any, will be unlocked. In this way, only the items that need to be updated will be securely locked. After releasing the unnecessary node-level locks on the nodes that do not need to be changed, if the cell contains the old location will not have any data object after the update, the corresponding entry in the leaf node will be deleted. If this leaf node has the capacity more than or equal to the minimum node capacity, the delete process is accomplished. Otherwise, a node merge operation similar to the concurrent merge in B^{link}-trees has to be performed to restructure the tree. In this way, the delete operation assures that other concurrent operations can obtain valid results. On the other hand, if the new location is in an empty cell, a new entry will be added to the corresponding leaf node. If this leaf node for insertion has sufficient space for a new entry, the cell will be added to this node directly, and then the write-lock will be released. Otherwise, the leaf node will be split into two nodes by adding a new leaf node as the right neighbor of the original node, following the concurrent split operation in B^{link}-trees. A propagation split will be performed if necessary, following the cached path. After modifying the B^{link}-tree and releasing all the node-level locks, the operation then updates the actual data pages.

The **final phase, commitment,** releases the write-locks on the cells and returns the updated B^{link}-tree. The detailed concurrent location update algorithm is described in Algorithm 1.

4.3 Range Query

Algorithm RangeQuery (*R*, *T*, *LM*)
Input: *R:* Query range, *T*: B^{link}-tree, *LM:* Lock map,
Output: *S*: Set of objects covered by *R*.

//Initiation
1. *S* = {}; // initiate the result set
2. *L*={}; //initiate locked set
3. *SC* = *SFC_map(R)*; //determine the cells overlap with *R* using SFC

//B^{link}-tree traversal
4. For each cell cluster *C* in *SC*
5. *n* = *T.traverse(C)*; // locate the left-most leaf node which overlaps with cell cluster *C*
6. While (*n.minKey* <= *C.maxKey*)
7. *P* = *n.entries* ∩ *C*;
8. *LM.readLock(P)*; //request read-lock on cell set *P*
9. *L* = *L* + *P*; // record the locks
10. *S* = *S* + *PageRetrieve(n.entry(P))*; // retrieve objects inside *P*

//Commitment
11. *S* = *S* ∩ *R*; // filter the objects outside of *R*
12. *LM.unReadLock(L)*; // release the cell-level locks
13. Return *S*;

Algorithm 2. Concurrent range query.

Given a search range *R*, a range query returns all the objects that are covered by *R*. This operation requires only read-locks. To execute a range query, the spatial query range *R* will be mapped to a set of one-dimensional ranges using an SFC. After that, these one-dimensional ranges will be queried on the B^{link}-tree. Each one-dimensional query is executed as the fundamental concurrent search operation on the B^{link}-tree, whereas the major difference is that all the cells that overlap with *R* will be read-locked before being scanned, regardless whether they are empty or not. For example, in the range query *R* shown in Figure 2, the read-locks will be placed on cell 32, 33, 34, 35, 52, and 53, before retrieving any of them. Therefore, the corresponding records in the lock map will be checked, and if the cell is available, its read-lock counter will be incremented; otherwise, the corresponding process *ID* will be inserted into the pending queue. All these read-locks will not be released until the entire range query is complete. This locking strategy assures that these cells will not be altered during the entire process of the spatial range query. The detailed concurrent range query algorithm is presented in Algorithm 2. In the **initiation** step (line 1-3), the cells overlapped with the query range will be identified. In the **tree traversal** stage (line 4-10), for each consecutive cell cluster (e.g., cell 32-35 in query *R*), the tree will be traversed from the root to leaf, and the corresponding data cells will be read-locked before retrieving the data pages. Once all the indexed cells that overlap with the query range have been accessed, in the **commitment** step, exact results are returned and all the requested cell locks are released.

5 CORRECTNESS OF CONCURRENCY

In the proposed concurrent spatial operations, the three requirements of concurrency control protocols, namely, serializable isolation, data consistency, and deadlock free, can be achieved. **Serializable Isolation -** The node-level locks isolate the concurrent location update operations, because these operations place node-level write-locks in a bottom-up manner on the B^{link}-tree when reconstruction is required. On the other hand, the cell-level locks serialize the concurrent update and search operations on data pages. Even though the isolated order of operations may not be exactly in the same sequence as they started, it is regarded as valid in concurrency control protocols because it is inevitable to suspend or restart some operations. **Data Consistency -** The spatial concurrent operations can be assured to securely retrieve valid results consistent to the current status. For instance, suppose a range query *R* and an update operation *U* occur simultaneously with cell *C* as the common resource, the results of *R* will reflect *U* only when the read-lock is placed after the write-lock on *C*, which means the new data in *C* is inserted into the dataset before *R* is accomplished. Furthermore, in case the reconstruction caused by *U* is performed while the traversal of *R* is in process, as described in the B^{link}-tree search algorithm, *R* can always follow the down-and-right links to reach the leaf node that currently contains *C*. The only situation that *U* will not impact the results of *R*, is that the write-lock on *C* is successfully placed after all the read-locks from *R* are released, which means the data in cells *C* is inserted after the commitment of *R*. Two concurrent update operations can guarantee the final results are consistent to the current dataset, because they can only be processed with the cell-level write-locks and bottom-up node-level locks successfully granted, which prevents any confliction. **Deadlock Free -** The proposed operations will not cause additional deadlocks, because range queries need to access multiple cells, and they only read-lock these cells. Meanwhile, each location update operations place cell-level write-locks at one time, which will not cause deadlocks with search operations. In an update operation, all node-level locks will be placed at one time, and these locks will either release, or expand upward or rightward during this process. Furthermore, each update operation will request cell-level locks after write-locking the corresponding leaf nodes. Therefore, there will not be any two update operations that hold the resources required by each other and pending indefinitely. A detailed proof by examining all possible combinations of these concurrent operations is given as follows.

Proof:

We only need to prove that a location update will not interfere with any concurrent operations, because a range query can never

affect any other range queries. There are two conditional branches in the proposed location update operation. Following each branch, a location update may occur simultaneously with a range query and two types of another location update on common cells.

Branch 1: Without index modification

In this branch, the cell that contains the old location will not be empty after the update, and the cell that will encompass the new location exists before the update. At first, node-level locks will be placed together at one time on leaf nodes *n_new* and *n_old*. Cell-level write-locks will then be requested together at one time on cell *c_old* and *c_new* using lock map before unlocking the leaf nodes and before updating the data pages. If a **range query** starts during this update and covers cell *c_old* or *c_new*, it needs to obtain read-lock on cell *c_old* or *c_new* before actually reading the data pages. Since this read-lock is exclusive to the write-locks requested by the location update, the range query will have to wait if the write-locks have been placed, or will keep the location update waiting if the write-locks have not been placed. Therefore, the intermediate status of the location update will not be retrieved. Furthermore, because the write-locks on cell *c_old* and *c_new* are requested together as an atomic action, the location update will either wait without holding any locks or proceed after obtaining all the requested locks. There will be no deadlock occurring during this process.

If another **location update** (say, *U**) in **branch 1** or **branch 2** starts during this update and affects cell *c_old* or *c_new*, it needs to request cell-level write-locks on cell *c_old* or *c_new* before actually modifying the data pages. Since these write-locks are exclusive to the write-locks requested by the original location update, *U** will have to wait if the write-locks have been placed by the original update, or will keep the original update waiting if the write-locks have not been placed. Therefore, inconsistent status of the index and data will not be retrieved. In this branch, all the cell-level locks/node-level locks in each operation are requested as an atomic action, so there will be no deadlock occurs. All other operations are not related to this location update from the aspect of concurrency control.

Branch 2: With index modification

The cell that contains the old location will be empty after the update, or the cell that contains the new location does not exist before the update. In this branch, the cell-level write-locks will be requested on cell *c_old* and *c_new*, and the node-level locks will be placed on the B^{link}-tree leaf nodes *n_old* and *n_new* at the beginning, and be kept on the nodes needed to be modified till the end of the operation. In case a **range query** starts during this update process and covers cell *c_old* or *c_new*, it needs to obtain read-lock on cell *c_old* or *c_new* before actually reading the data pages. Since this read-lock is exclusive to the write-locks requested by the location update, the range query will have to wait if the write-locks have been placed, or will keep the location update pending if the write-locks have not been placed. Therefore, the intermediate status of the location update will not be accessed. Furthermore, because the write-locks on cell *c_old* and *c_new* are requested together as an atomic action, the location update will either wait without holding any locks or proceed after obtaining all the requested locks. There will be no deadlock occurred during this process.

If another **location update** *U** in **branch 1** starts during this update and affects cell *c_old* or *c_new*, it needs to request cell-level write-locks on cell *c_old* or *c_new* before actually modifying the data pages. Since these write-locks are exclusive to the write-locks requested by the original location update, *U** will have to wait if the write-locks have been placed by the original update, or will keep the original update waiting if the write-locks have not been placed. Therefore, inconsistent status of the index and data will not be retrieved. Similarly, since all the write-locks in each operation are requested as an atomic action, no deadlock will occur in this case. If *U** is a **location update** in **branch 2** that starts during this update and affects tree node *n_old* or *n_new*, it needs to request node-level locks on *n_old* or *n_new* before requesting cell-level locks and before actually modifying the tree nodes and data pages. Since these locks are exclusive to the node-level locks requested by the original location update, *U** will have to wait if the node-level locks have been placed by the original update, or will keep the original update waiting if these locks have not been placed. Therefore, inconsistent status of the index and data will not be retrieved. Similarly, since all the node-level locks or cell-level locks in each operation are requested together as an atomic action, no deadlock will occur in this branch. Furthermore, as the cell-level locks are placed after the node-level locks have been obtained, if any location update needs to modify the corresponding leaf node, it can always retrieve the valid nodes. All other operations are not related to this location update from the aspect of concurrency control.

To summarize the proof, all the possible combinations of concurrent spatial operations and their conditional braches have been thoroughly examined. All of them are shown to meet the requirements of serializable isolation, data consistency, and deadlock free. This proof is complete.

6 EXPERIMENTS

To evaluate the performance of the proposed concurrent spatial operations, sets of experiments on real datasets have been conducted by comparing the throughputs (number of operations processed in a time unit) among different concurrency control protocols, as shown in Figure 4. The two real datasets used in the experiments are 6,000 road network nodes in the city of Oldenburg and 62,000 points of interest in California, both from [13], as shown in Figure 5. The road nodes for road network in the city of Oldenburg, mapped using a Hilbert curve with order 5, are relatively uniformly distributed with about 40% of empty cells. On the other hand, the dataset for points of interest in California, mapped using a Hilbert curve with order 8, has a rather skew distribution with about 80% of empty cells. Based on the SFCs, a B^{link}-tree with height of 3 was built on the city of Oldenburg dataset, and a tree with height of 4 on the California dataset, respectively. The fanout of the B^{link}-trees built in experiments is 32.

In the experiments, the range queries were created by randomly selecting the center points and setting the window size as 5% of the whole data spaces, and each location update was generated by randomly assigning an existing object as a start point and half length of the cell width as the moving pace. The **concurrency level** (the number of operations simultaneously processed in one batch) of the operations, and the **mobility rate** (the percentage of location updates in the entire operation set) of the moving objects varied in the experiments as the parameters to simulate different application environments. The processing time of operations was used to evaluate the performance of the proposed concurrent operation framework. A **fusion** concurrency control approach, which applies the link-based locking on the B^{link}-tree and the lock-

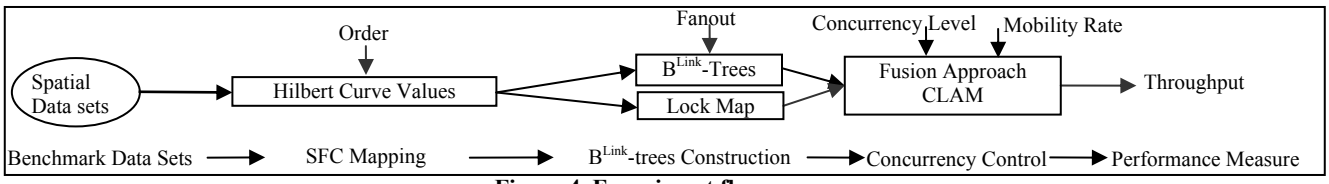
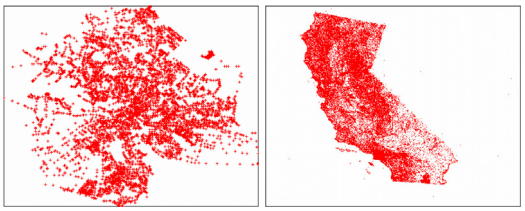


Figure 4. Experiment flow.

coupling locking on the lock map, was designed and implemented in the experiments for comparing with the proposed concurrent protocol. Different from CLAM, this fusion protocol requests the locks at the beginning of a location update and releases them in the end. This fusion approach applies the link-based locking on the index tree, which has been shown to have fewer number of read/write conflictions with less maintenance overhead than lock-coupling protocols, therefore it can achieve higher throughput than the pure lock-coupling approach. Note that the proposed approach was compared to a non-trivial method to demonstrate the advantage against an advanced method.

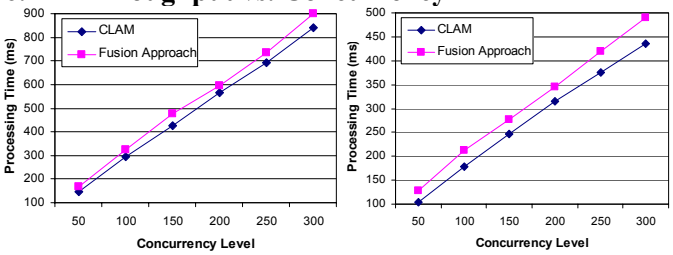


(a) City of Oldenburg. (b) California places.

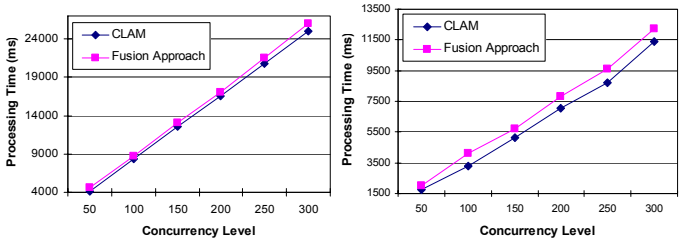
Figure 5. Experiment datasets.

Two sets of experiments are described in the following subsections. The first set of experiments shows the efficiency and scalability of the proposed location management on Hilbert curves by examining the throughputs under different concurrency levels. The second set of experiments demonstrates the impact of mobility rate on throughputs. The comparisons between the fusion concurrency control approach and the proposed CLAM framework were made in both sets of experiments.

6.1 Throughput vs. Concurrency



a) Oldenburg data (with mobility rate = 30 (left) and 70 (right)).



b) California data (with mobility rate = 30 (left) and 70 (right)).

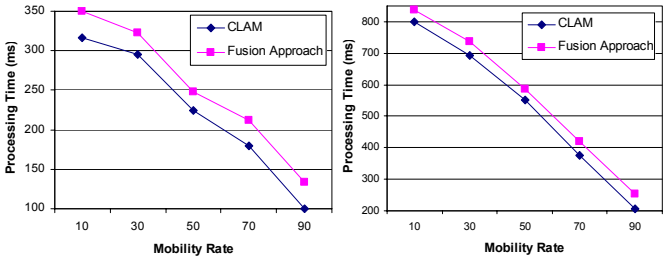
Figure 6. Processing time under different concurrency levels.

This set of experiments compares the processing time between concurrent location management of CLAM and the fusion approach under different concurrency levels, in order to determine how the concurrency workload affects the system throughput. The processing time of the concurrent operations was collected with the mobility rate sets as 30 percent and 70 percent of the whole operation set. Similar trends of the processing time have also been observed under different mobility rates.

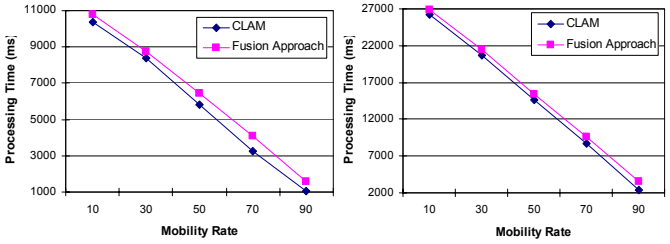
Figure 6 shows the processing time of the concurrent operations with various concurrency levels, in which the X-axis represents the concurrency levels and the Y-axis indicates the processing time in milliseconds. As illustrated in Figure 6, in both datasets, the processing time of both concurrency control approaches generally increases proportionally with the concurrency level. In this set of experiments, CLAM performs 10-20% better than the fusion approach. Furthermore, the higher the concurrency level, the larger the gap between these two approaches. This is reasonable because more concurrent operations in a batch could cause more read-write conflicts. Consequently, the processing time saved by efficiently releasing unnecessary locks, as employed in CLAM, will become more significant.

6.2 Throughput vs. Mobility

This set of experiments compares the processing time between CLAM and the fusion concurrency control approach with different mobility rates. The processing time of the concurrent operations was collected with the concurrency level set at 100 and 250 correspondingly. Similar trends of the processing time have been observed under different concurrency levels.



a) Oldenburg Data (with concurrency = 100 (left) and 250 (right)).



b) California data (with concurrency = 100 (left) and 250 (right)).

Figure 7. Processing time under different mobility rates.

Figure 7 shows the processing time of the concurrent operations as the mobility rate increases, where the X-axis indicates the mobility rates and the Y-axis represents the processing time in milliseconds. As observed from Figure 7, in both datasets, the processing time of both approaches linearly decreases when the mobility rate

increases. This is because a range query usually needs to access as many data pages as the number of SFC cells it covers, while a location update only needs to access at most two data pages. In this set of experiments, CLAM performs significantly better than the fusion approach. Furthermore, the advantage of CLAM against the fusion approach becomes more significant when the mobility rate increases. For example, in the Oldenburg dataset with the concurrency level of 100 (left figure in Figure 7(a)), the fusion approach takes 10% longer than CLAM to process the operations when the mobility rate is 10, while it takes 30% longer time than CLAM when the mobility rate is 90. This is because that CLAM optimizes the locking strategy in the location update operation (Algorithm 1). When there are more location update operations, CLAM is expected to benefit more from its optimizations.

These experiment results show the proposed concurrent location management approach exhibits scalable performance when the concurrency level increases or when the mobility rate decreases. It outperforms the fusion concurrency control approach under different scenarios, especially in the high mobility situation. This indicates that the proposed concurrency framework can achieve prominent performance in general, and is suitable to manage frequent concurrent location updates and queries.

7 CONCLUSION & FUTURE WORKS

This paper proposes a concurrent location management framework, CLAM, for efficiently handling moving objects. CLAM provides adequate protection for concurrent location update and search operations to achieve serializability, consistency, and deadlock free. The correctness of CLAM is formally proved by completely examining all the possible scenarios during the concurrent operation processing. Experiment results on real datasets have validated that the performance optimizations in CLAM are effective. Further efforts could focus on extending CLAM to support other spatial operations, such as the range aggregation and the nearest neighbor search. Meanwhile, adopting the design of CLAM to other moving object access methods based on B+-trees offers further attractive possibilities.

8 REFERENCES

[1] Abiteboul, S., Agrawal, R., Bernstein, P., et al. 2005. The Lowell Database Research Self-Assessment. Commun. ACM 48, 5 (May. 2005), 111-118.

[2] Beckmann, N., Kriegel, H. P., Schneider, R., et al. 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In Proceedings of ACM SIGMOD International Conference on Management of Data, (Atlantic City, NJ, USA, May 23-25, 1990). 322-331.

[3] Faloutsos, C. 1986. Multiattribute Hashing Using Gray Codes. In Proceedings of ACM SIGMOD International Conference on Management of Data, (Washington, D.C., USA, May 28-30, 1986). 227-238.

[4] Gaede, V. and Gunther, O. 1998. Multidimensional Access Methods. ACM Comput. Surv. 30, 2 (Jun. 1998), 170-231.

[5] Guttman, A. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In Proceedings of ACM SIGMOD International Conference on Management of Data, (Boston, MA, USA, June 18-21, 1984). 47-57.

[6] Hilbert, D. 1981. Ueber Stetige Abbildung Einer Linie Auf Ein Flashenstuck. Mathematishe Annalen, (1981), 459-460.

[7] Jagadish, H. V. 1990. Linear Clustering of Objects with Multiple Attributes. In Proceedings of ACM SIGMOD International Conference on Management of Data (Atlantic City, NJ, USA, May 23-25, 1990). 332 - 342.

[8] Jaluta, I., Sippu, S. and Soisalon-Soininen, E. 2005. Concurrency Control and Recovery for Balanced B-link Trees. VLDB J. 14, 2 (Apr. 2005), 257-277.

[9] Jonge, W. d. and Schiff, A. 1990. Concurrent Access to B-trees. In Proceedings of PARBASE International Conference on Databases, Parallel Architectures and Their Applications, (Miami Beach, FL, USA, March 7-9, 1990). 312-320.

[10] Lanin, V. and Shasha, D. 1986. A Symmetric Concurrent B-tree Algorithm. In Proceedings of ACM Fall Joint Computer Conference, (Dallas, TX, USA, November 2-6, 1986). 380-389.

[11] Lawder, J. K. and King, P. J. H. 2001. Querying Multi-dimensional Data Indexed Using the Hilbert Space-Filling Curve. SIGMOD Record 30, 1 (Mar. 2001), 19-24.

[12] Lehman, P. and Yao, S. 1981. Efficient Locking for Concurrent Operations on B-trees. ACM T. Database Syst. 6, 4 (Dec. 1981), 650-670.

[13] Li, F. and Kollios, G. Real Datasets for Spatial Databases: Road Networks and Points of Interest. from http://cs-people.bu.edu/lifeifei/SpatialDataset.htm, Last Accessed on May 2, 2007.

[14] Liao, S., Lopez, M. A. and Leutenegger, S. T. 2001. High Dimensional Similarity Search with Space Filling Curves. In Proceedings of the 17th IEEE International Conference on Data Engineering, (Heidelberg, Germany, April 2-6, 2001). 615-622.

[15] Mokbel, M. F., Aref, W. G. and Kamel, I. 2003. Analysis of Multi-dimensional Space-Filling Curves. Geoinformatica 7, 3 (Sep. 2003), 179-209.

[16] Moon, B., Jagadish, H. V., Faloutsos, C., et al. 2001. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. IEEE T. Knowl. Data En. 13, 1 (Jan./Feb. 2001), 124-141.

[17] Nievergelt, J., Hinterberger, H. and Sevcik, K. C. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM T. Database Syst. 9, 1 (Mar. 1984), 38-71.

[18] Orenstein, J. A. 1986. Spatial Query Processing in an Object-Oriented Database System. In Proceedings of ACM SIGMOD International Conference on Management of Data (Washington, D.C., USA, May 28-30, 1986). 326-336.

[19] Peano, G. 1890. Sur Une Courbe Qui Remplit Toute Une Air Plaine. Math. Ann. 36, (1890), 157-160.

[20] Ramakrishnan, R. and Gehrke, J. 2001. Database Management Systems. McGraw-Hill, New York, NY, USA.

[21] Sagan, H. 1994. Space Filling Curves. Springer, Berlin, Germany.

[22] Setzer, V. W. and Zisman, A. 1994. New Concurrency Control Algorithms for Accessing and Compacting B-trees. In Proceedings of the 20th International Conference on Very Large Data Bases, (Santiago de Chile, Chile, September 12-15, 1994). 238-248.